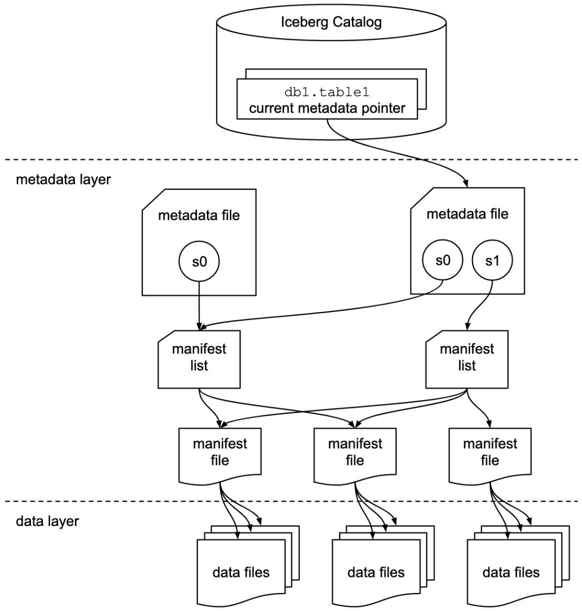
Catalog/Datasource

背景回顾

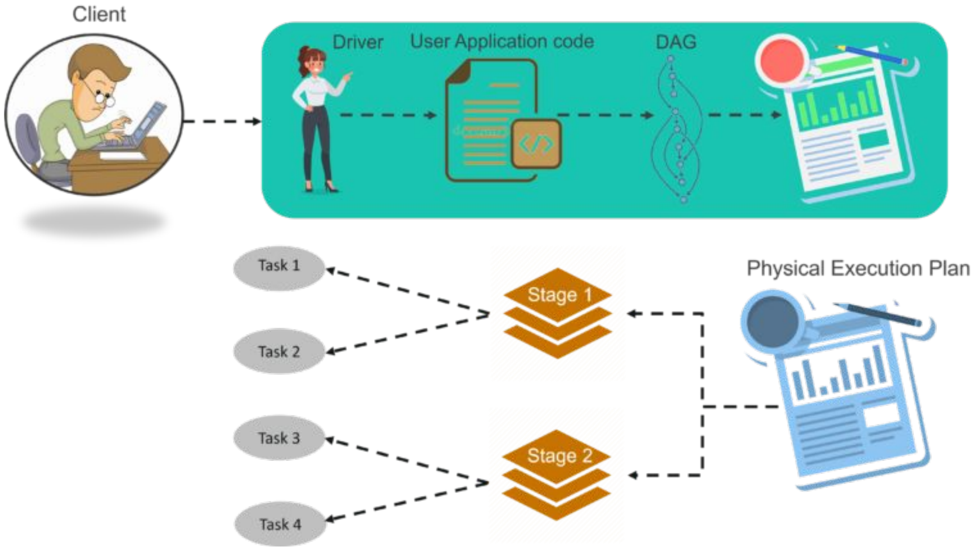
多版本的表



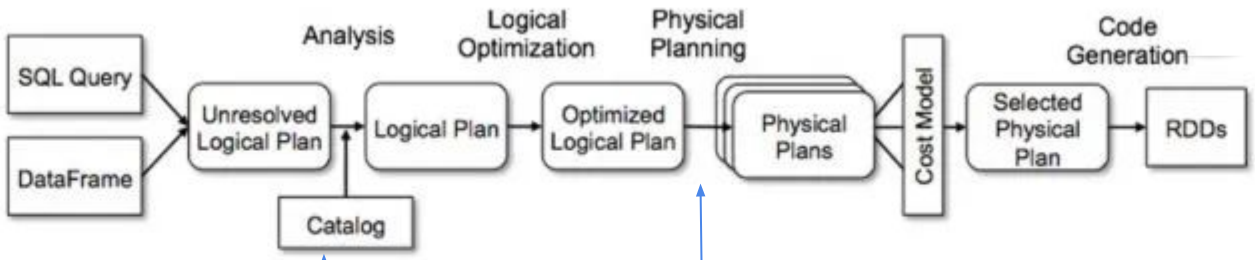
Shamelessly copied from Iceberg

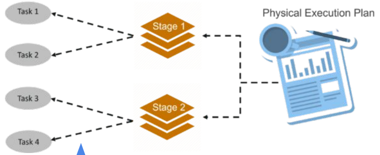
背景回顾

分布式地执行


Client
Driver
User Application code
DAG
Physical Execution Plan
Stage 1
Stage 2
Task 1
Task 2
Task 3
Task 4

现状





Datasource::get_table

Table::read_plan

tbl <- Datasource::get_table // by name

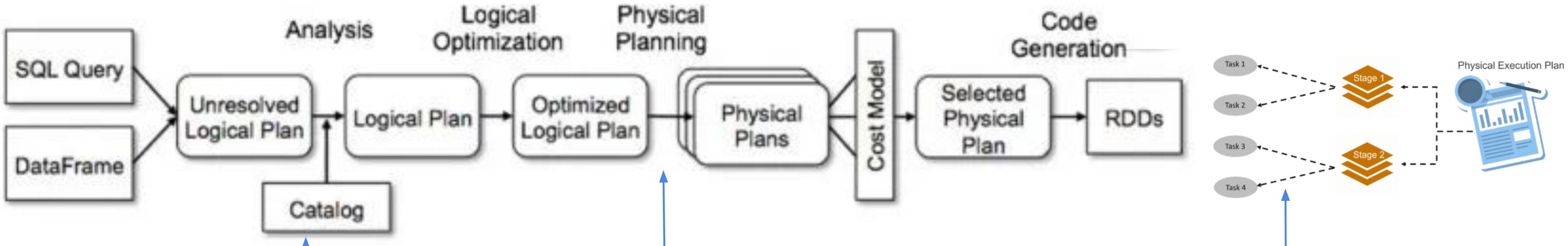
tbl.read

各节点，可能工作在不同的Table版本上

当前的改造

TableMeta 携带id和version

注：这部分与具体的事务并发控制机制有关



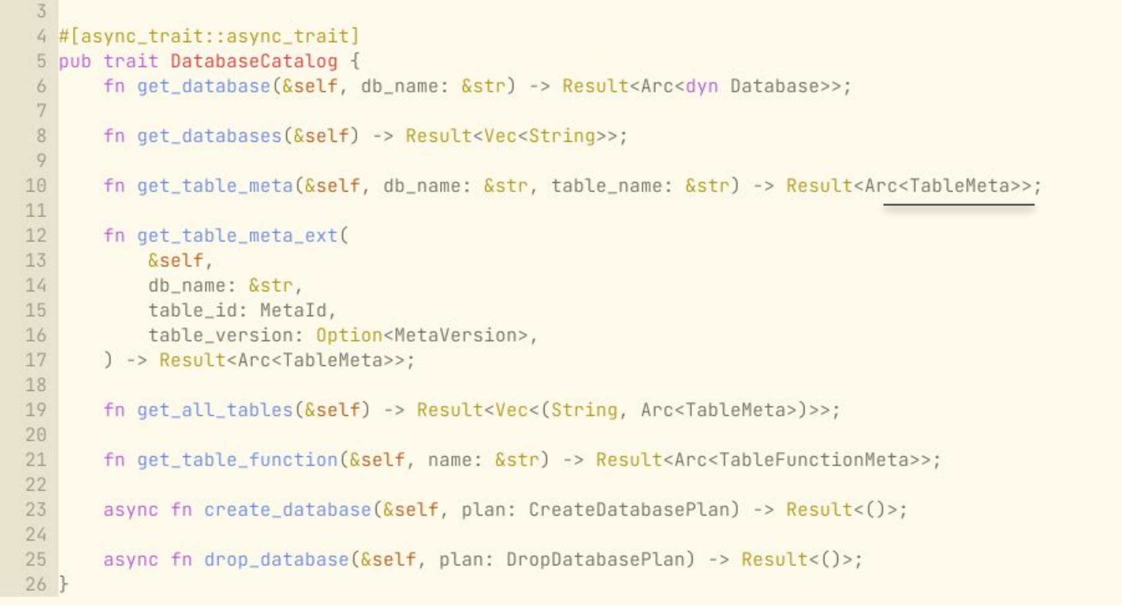
tble_meta.datasource().read_plan(...)

tbl_meta <- catalog.get_table_meta(db_name, tbl_name)

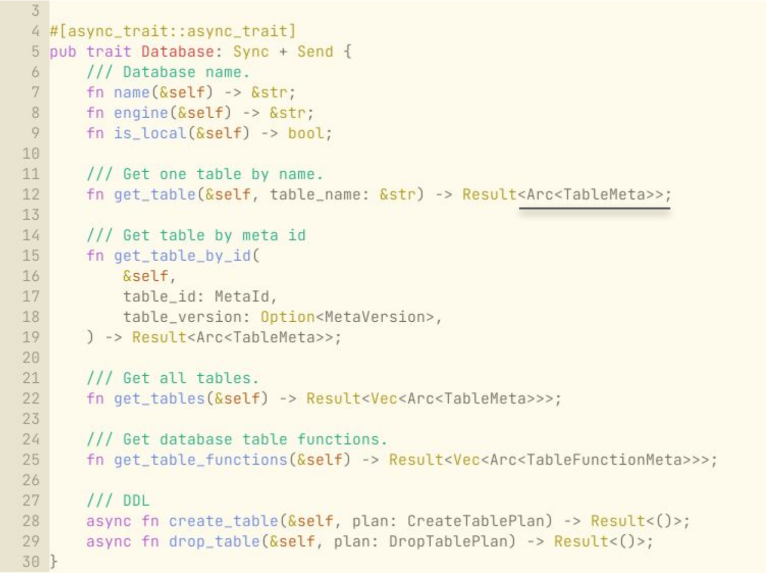
tbl_meta <-catalog.get_table_meta_ext(**meta_id,meta_ver**)

tbl_meta.datasource().read(..)

改造

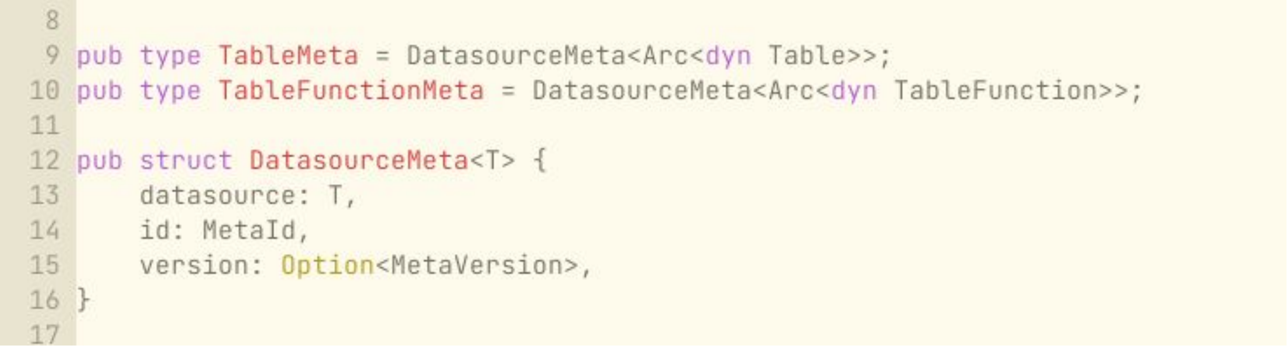


```rust
#[async_trait::async_trait]
pub trait DatabaseCatalog {
    fn get_database(&self, db_name: &str) -> Result<Arc<dyn Database>>;

    fn get_databases(&self) -> Result<Vec<String>>;

    fn get_table_meta(&self, db_name: &str, table_name: &str) -> Result<Arc<TableMeta>>;

    fn get_table_meta_ext(
        &self,
        db_name: &str,
        table_id: MetaId,
        table_version: Option<MetaVersion>,
    ) -> Result<Arc<TableMeta>>;

    fn get_all_tables(&self) -> Result<Vec<(String, Arc<TableMeta>)>>;

    fn get_table_function(&self, name: &str) -> Result<Arc<TableFunctionMeta>>;

    async fn create_database(&self, plan: CreateDatabasePlan) -> Result<()>;

    async fn drop_database(&self, plan: DropDatabasePlan) -> Result<()>;
}
```

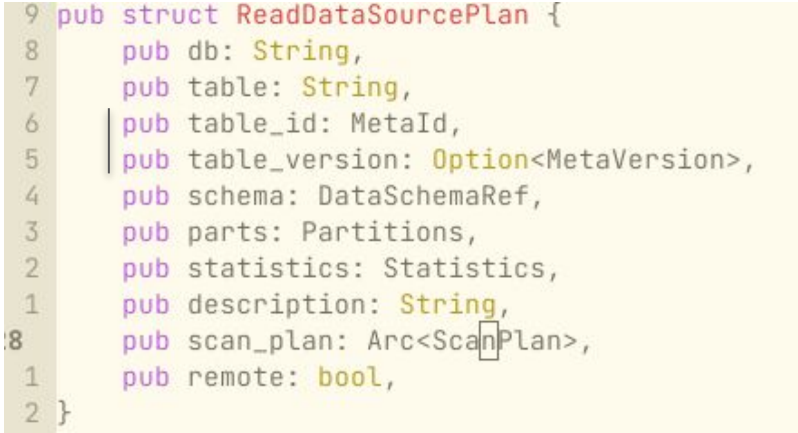


```rust
#[async_trait::async_trait]
pub trait Database: Sync + Send {
    /// Database name.
    fn name(&self) -> &str;
    fn engine(&self) -> &str;
    fn is_local(&self) -> bool;

    /// Get one table by name.
    fn get_table(&self, table_name: &str) -> Result<Arc<TableMeta>>;

    /// Get table by meta id
    fn get_table_by_id(
        &self,
        table_id: MetaId,
        table_version: Option<MetaVersion>,
    ) -> Result<Arc<TableMeta>>;

    /// Get all tables.
    fn get_tables(&self) -> Result<Vec<Arc<TableMeta>>>;

    /// Get database table functions.
    fn get_table_functions(&self) -> Result<Vec<Arc<TableFunctionMeta>>>;

    /// DDL
    async fn create_table(&self, plan: CreateTablePlan) -> Result<()>;
    async fn drop_table(&self, plan: DropTablePlan) -> Result<()>;
}
```

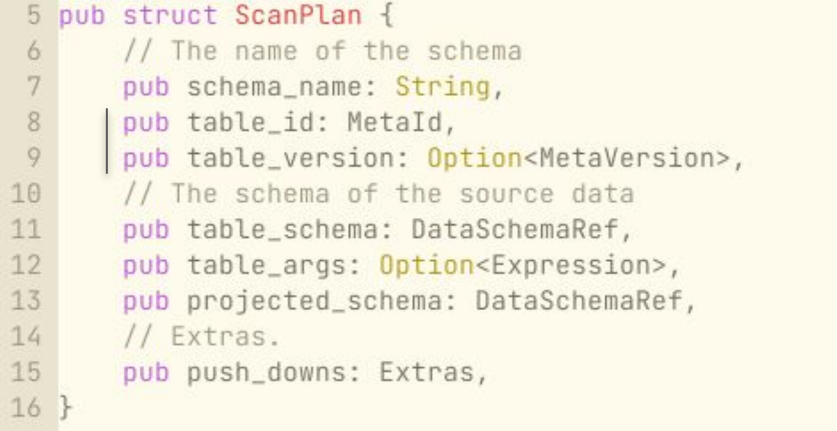
改造



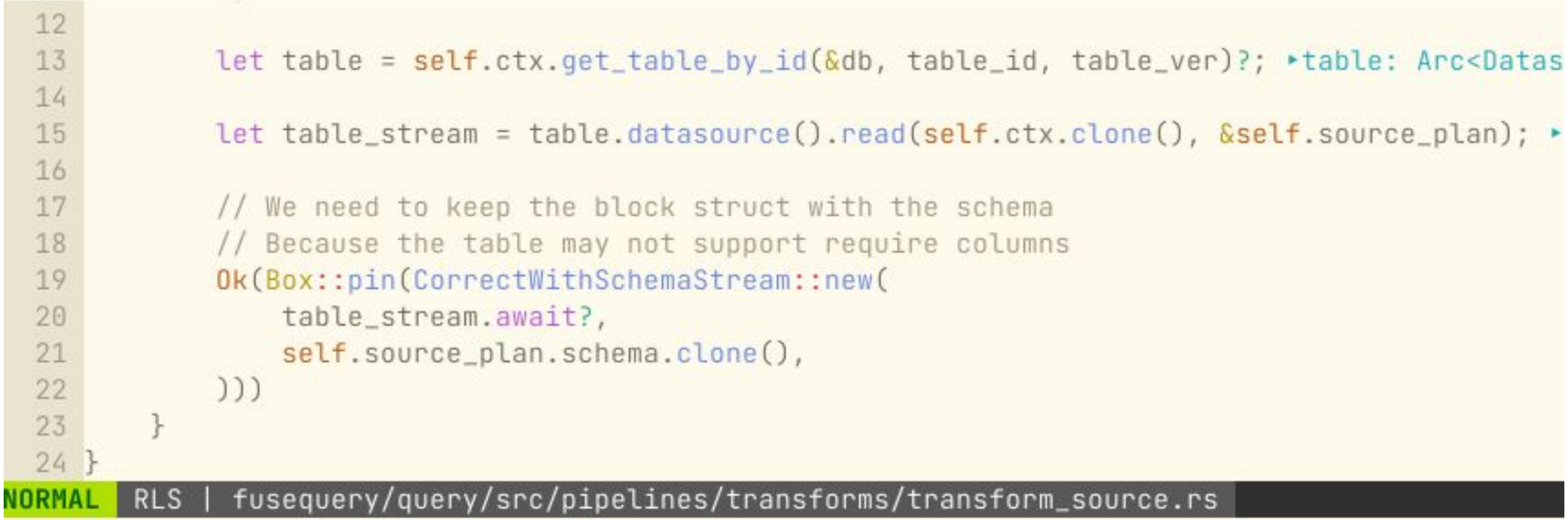
```rust

pub type TableMeta = DatasourceMeta<Arc<dyn Table>>;
pub type TableFunctionMeta = DatasourceMeta<Arc<dyn TableFunction>>;

pub struct DatasourceMeta<T> {
    datasource: T,
    id: MetaId,
    version: Option<MetaVersion>,
}

```



```rust
pub struct ReadDataSourcePlan {
    pub db: String,
    pub table: String,
    pub table_id: MetaId,
    pub table_version: Option<MetaVersion>,
    pub schema: DataSchemaRef,
    pub parts: Partitions,
    pub statistics: Statistics,
    pub description: String,
    pub scan_plan: Arc<ScanPlan>,
    pub remote: bool,
}
```



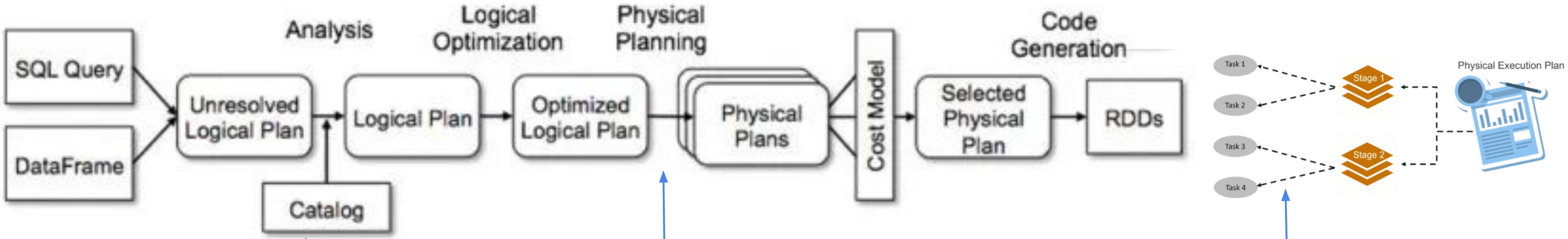
```rust
pub struct ScanPlan {
    // The name of the schema
    pub schema_name: String,
    pub table_id: MetaId,
    pub table_version: Option<MetaVersion>,
    // The schema of the source data
    pub table_schema: DataSchemaRef,
    pub table_args: Option<Expression>,
    pub projected_schema: DataSchemaRef,
    // Extras.
    pub push_downs: Extras,
}
```



```

        let table = self.ctx.get_table_by_id(&db, table_id, table_ver)?; ▸table: Arc<Datas

        let table_stream = table.datasource().read(self.ctx.clone(), &self.source_plan); ▸

        // We need to keep the block struct with the schema
        // Because the table may not support require columns
        Ok(Box::pin(CorrectWithSchemaStream::new(
            table_stream.await?,
            self.source_plan.schema.clone(),
        )))
    }
}
```

NORMAL RLS | fusequery/query/src/pipelines/transforms/transform_source.rs

Ideally



tble_meta.datasource().read_plan(...)

tbl_meta <- catalog.get_table_meta(db_name, tbl_name)

- Table Meta已知
- 不再依赖Catalog
- 仅与**狭义**的Storage交互
- 需要进一步剥离Datasource

Catalog和MetaStore

- (DB)MetaStore
  - Data Dictionary
  - Store层组件，维护DB元数据，并对外提供RPC服务
- Catalog
  - Query层组件，MetaStore的client

Catalog和MetaStore的同步

- 定期同步
    - 通过DB Meta Version，判断是否有更新
    - 目前仅有全量更新接口
    - 明显的问题——不新鲜
        - 分布式执行过程中，早晚会出 问题
    - 其它
        - 全量同步较重
        - 增量同步与RSM能力重复，或可复用
        - Doris/TiDB的思路(不完全适合咱们)

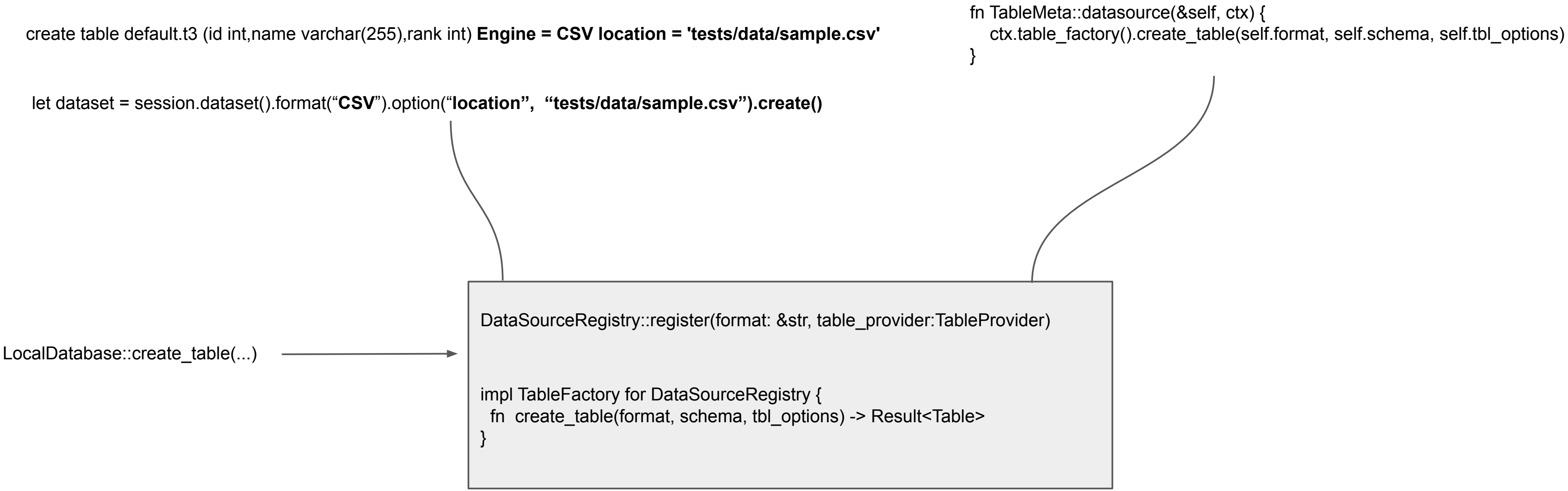
注：read_plan在store层执行

Catalog和MetaStore的同步

- 仅缓存访问过的TableMeta
  - 给定(id, version)的TableMeta是immutable的
  - 仅get_table_meta_ext(id, ver)中使用cache
  - LRU Cache (Read-Through)
- get_table_meta(db_name, tbl_name)
  - 每次都通过rpc获取，并cache结果(带版本信息的)

Datasource

Almost there



```
create table default.t3 (id int,name varchar(255),rank int) Engine = CSV location = 'tests/data/sample.csv'
```

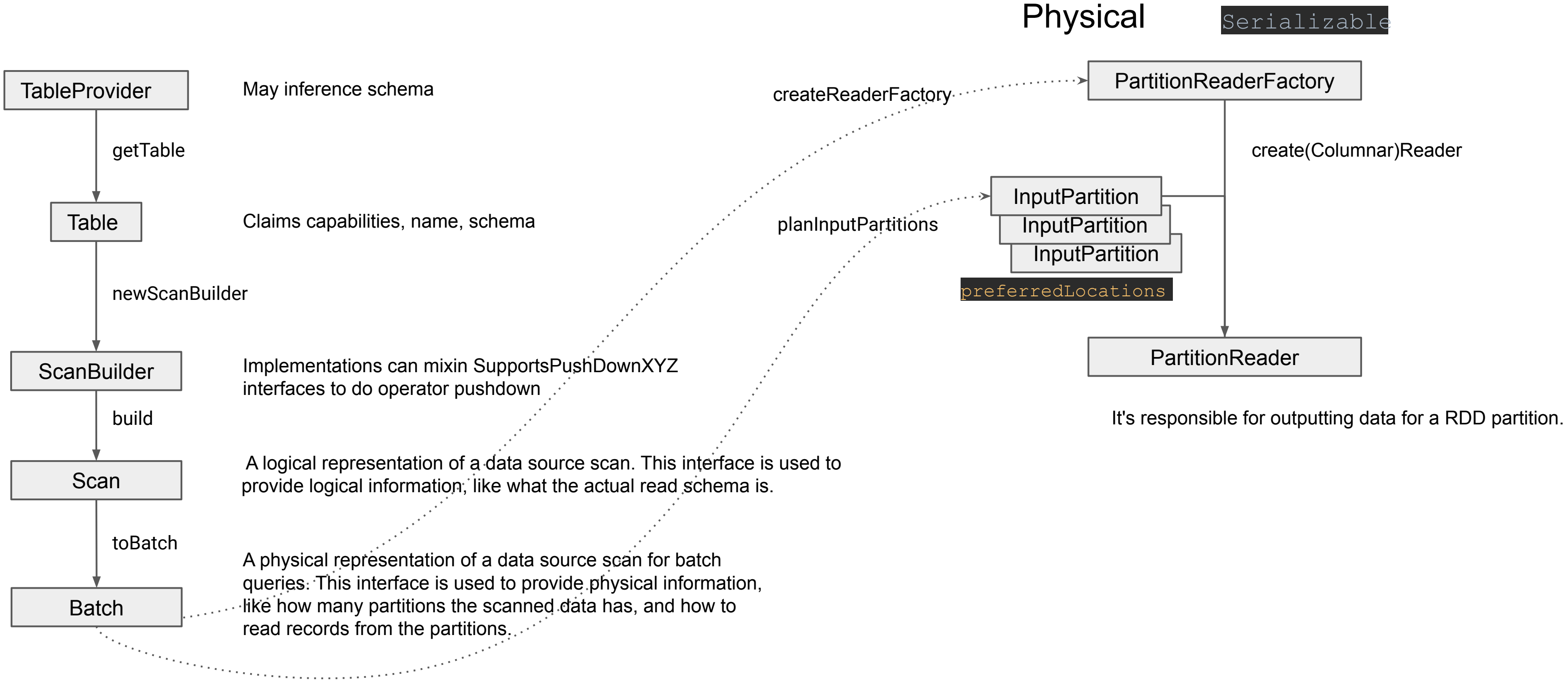
```
let dataset = session.dataset().format(“CSV”).option(“location”,  “tests/data/sample.csv”).create()
```

```
fn TableMeta::datasource(&self, ctx) {
    ctx.table_factory().create_table(self.format, self.schema, self.tbl_options)
}
```

```
LocalDatabase::create_table(...)
```

```
DataSourceRegistry::register(format: &str, table_provider:TableProvider)


impl TableFactory for DataSourceRegistry {
  fn  create_table(format, schema, tbl_options) -> Result<Table>
}
```

Datasource read



May inference schema

Claims capabilities, name, schema

Implementations can mixin SupportsPushDownXYZ interfaces to do operator pushdown

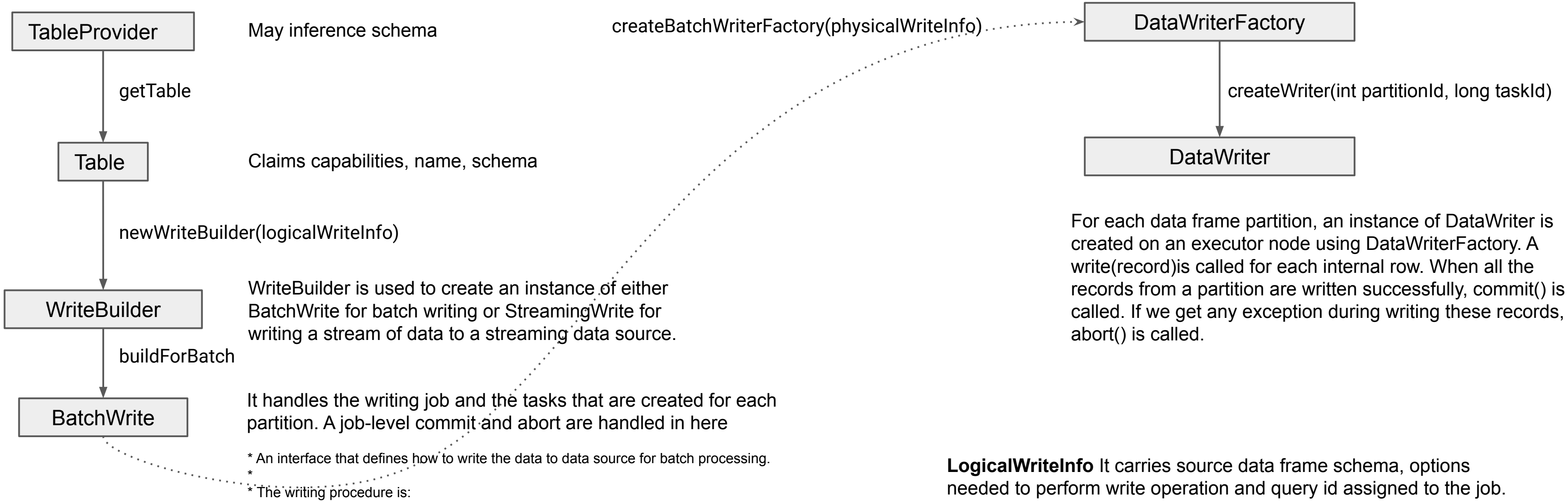
A logical representation of a data source scan. This interface is used to provide logical information, like what the actual read schema is.

A physical representation of a data source scan for batch queries. This interface is used to provide physical information, like how many partitions the scanned data has, and how to read records from the partitions.

It's responsible for outputting data for a RDD partition.

Easy Guide to Create a Custom Read Data Source in Apache Spark 3
https://levelup.gitconnected.com/easy-guide-to-create-a-custom-read-data-source-in-apache-spark-3-194afdc9627a

Datasource write



Physical

TableProvider

May inference schema

getTable

Table

Claims capabilities, name, schema

newWriteBuilder(logicalWriteInfo)

WriteBuilder

WriteBuilder is used to create an instance of either BatchWrite for batch writing or StreamingWrite for writing a stream of data to a streaming data source.

buildForBatch

BatchWrite

It handles the writing job and the tasks that are created for each partition. A job-level commit and abort are handled in here

createBatchWriterFactory(physicalWriteInfo)

DataWriterFactory

createWriter(int partitionId, long taskId)

DataWriter

For each data frame partition, an instance of DataWriter is created on an executor node using DataWriterFactory. A write(record)is called for each internal row. When all the records from a partition are written successfully, commit() is called. If we get any exception during writing these records, abort() is called.

```
* An interface that defines how to write the data to data source for batch processing.
*
* The writing procedure is:
*   1. Create a writer factory by {@link #createBatchWriterFactory(PhysicalWriteInfo)}, serialize
*      and send it to all the partitions of the input data(RDD).
*   2. For each partition, create the data writer, and write the data of the partition with this
*      writer. If all the data are written successfully, call {@link DataWriter#commit()}. If
*      exception happens during the writing, call {@link DataWriter#abort()}.
*   3. If all writers are successfully committed, call {@link #commit(WriterCommitMessage[])}. If
*      some writers are aborted, or the job failed with an unknown reason, call
*      {@link #abort(WriterCommitMessage[])}.
```

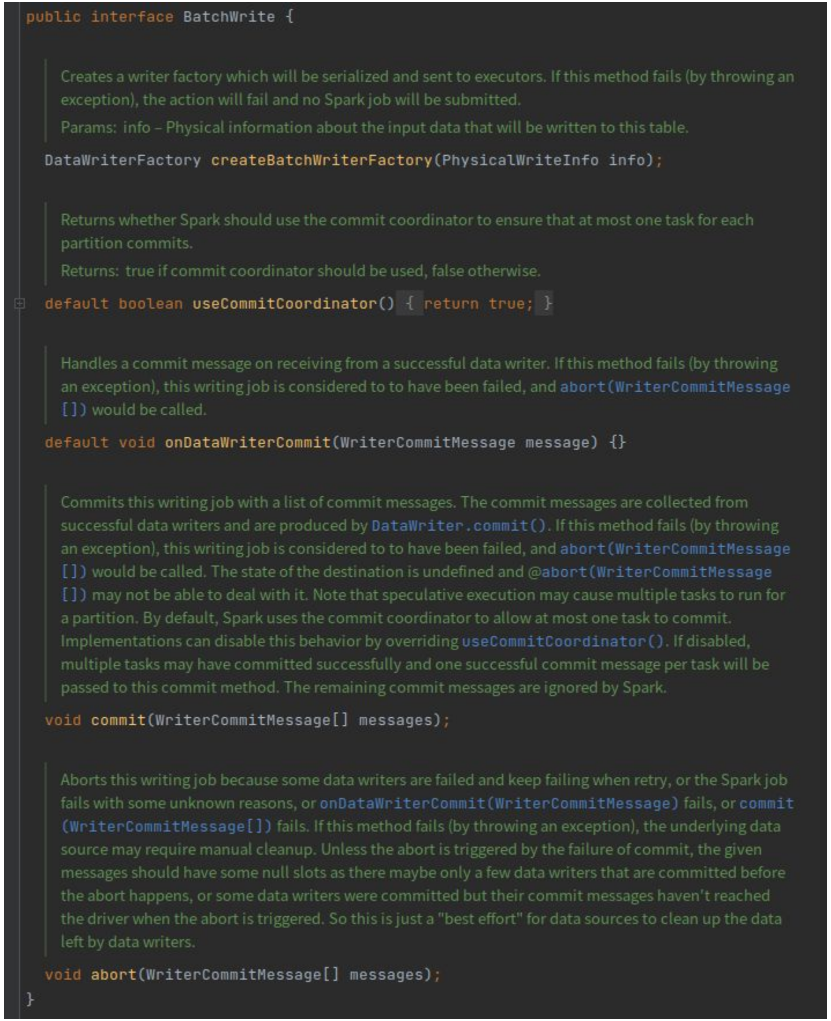
LogicalWriteInfo It carries source data frame schema, options needed to perform write operation and query id assigned to the job.

Easy Guide to Create a Custom Write Data Source in Apache Spark 3
https://levelup.gitconnected.com/easy-guide-to-create-a-write-data-source-in-apache-spark-3-f7d1e5a93bdb

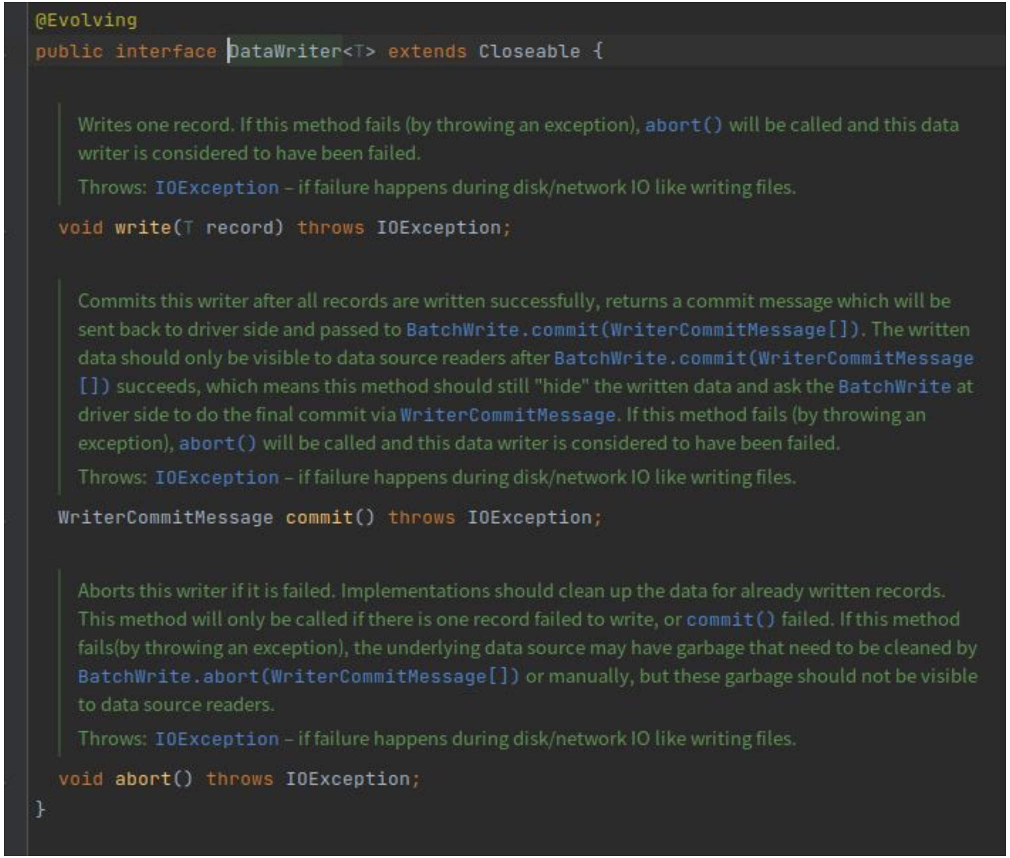
backups

其它

- Column id
  - Alter table rename column
- Database id
  - Alter db rename
- etc.



```
public interface BatchWrite {

    Creates a writer factory which will be serialized and sent to executors. If this method fails (by throwing an
    exception), the action will fail and no Spark job will be submitted.
    Params: info – Physical information about the input data that will be written to this table.
  DataWriterFactory createBatchWriterFactory(PhysicalWriteInfo info);

    Returns whether Spark should use the commit coordinator to ensure that at most one task for each
    partition commits.
    Returns: true if commit coordinator should be used, false otherwise.
  default boolean useCommitCoordinator() { return true; }

    Handles a commit message on receiving from a successful data writer. If this method fails (by throwing
    an exception), this writing job is considered to to have been failed, and abort(WriterCommitMessage
    []) would be called.
  default void onDataWriterCommit(WriterCommitMessage message) {}

    Commits this writing job with a list of commit messages. The commit messages are collected from
    successful data writers and are produced by DataWriter.commit(). If this method fails (by throwing
    an exception), this writing job is considered to to have been failed, and abort(WriterCommitMessage
    []) would be called. The state of the destination is undefined and @abort(WriterCommitMessage
    []) may not be able to deal with it. Note that speculative execution may cause multiple tasks to run for
    a partition. By default, Spark uses the commit coordinator to allow at most one task to commit.
    Implementations can disable this behavior by overriding useCommitCoordinator(). If disabled,
    multiple tasks may have committed successfully and one successful commit message per task will be
    passed to this commit method. The remaining commit messages are ignored by Spark.
  void commit(WriterCommitMessage[] messages);

    Aborts this writing job because some data writers are failed and keep failing when retry, or the Spark job
    fails with some unknown reasons, or onDataWriterCommit(WriterCommitMessage) fails, or commit
    (WriterCommitMessage[]) fails. If this method fails (by throwing an exception), the underlying data
    source may require manual cleanup. Unless the abort is triggered by the failure of commit, the given
    messages should have some null slots as there maybe only a few data writers that are committed before
    the abort happens, or some data writers were committed but their commit messages haven't reached
    the driver when the abort is triggered. So this is just a "best effort" for data sources to clean up the data
    left by data writers.
  void abort(WriterCommitMessage[] messages);
}
```



```
@Evolving
public interface DataWriter<T> extends Closeable {

    Writes one record. If this method fails (by throwing an exception), abort() will be called and this data
    writer is considered to have been failed.
    Throws: IOException – if failure happens during disk/network IO like writing files.
  void write(T record) throws IOException;

    Commits this writer after all records are written successfully, returns a commit message which will be
    sent back to driver side and passed to BatchWrite.commit(WriterCommitMessage[]). The written
    data should only be visible to data source readers after BatchWrite.commit(WriterCommitMessage
    []) succeeds, which means this method should still "hide" the written data and ask the BatchWrite at
    driver side to do the final commit via WriterCommitMessage. If this method fails (by throwing an
    exception), abort() will be called and this data writer is considered to have been failed.
    Throws: IOException – if failure happens during disk/network IO like writing files.
  WriterCommitMessage commit() throws IOException;

    Aborts this writer if it is failed. Implementations should clean up the data for already written records.
    This method will only be called if there is one record failed to write, or commit() failed. If this method
    fails(by throwing an exception), the underlying data source may have garbage that need to be cleaned by
    BatchWrite.abort(WriterCommitMessage[]) or manually, but these garbage should not be visible
    to data source readers.
    Throws: IOException – if failure happens during disk/network IO like writing files.
  void abort() throws IOException;
}
```

Table::append_data

T1	T2
> truncate table t1;	>
> begin;	>begin;
>	> select count(*) from t1; // returns 0
> insert into t1 values(...)	>
> commit;	>
	> select count(*) from t1; // returns c where c > 0

READ COMMITTED

foreach statement, grab a **fresh** TableMeta

Table::append_data

```
T1                                T2

> truncate table t1;              > begin;

> begin;                          >

>                                 > select count(*) from t1; // returns n

> insert into t1 values(...)      >

> commit;                         >

                                  >  select count(*) from t1; // returns n
```

REPEATABLE READ / SI

foreach tx, stick to a specific TableMeta

or mv-to

Table::append_data

T1

```
> begin;

> truncate table t1;

>

> insert into t1 values(...)

> select count(*) from t1; //returns n where n> 0
```

visibility